THE BELL SYSTEM TECHNICAL JOURNAL
Vol. 59, No. 9, November 1980
Printed in U.S.A.

Designer's Workbench:

Providing a Production Environment

By T. J. THOMPSON

(Manuscript received May 1, 1980)

In the development of Designer's Workbench, special consideration was given to the design, implementation, and maintenance of the production environment. Since DWB filled an immediate need in the computer-aided design world, production use of the system was started early and became an important factor as further development was planned. The reliability of the production system needed to be insured, while at the same time allowing the frequent changes and additions required in the normal course of development and general maintenance. To fulfill these requirements, we have implemented a "version control" system, allowing several independent versions of the entire system to be available simultaneously. In the general maintenance of DWB, we have used the Source Code Control System and make utility provided by the PWB/UNIX interactive operating system. We also have made extensive use of command procedures written in the UNIX "shell" language to perform the librarian and maintenance functions that are necessary to support the system.*

I. INTRODUCTION

Designer's Workbench is a software system based on the UNIX time-sharing operating system[1,2] which provides an interactive front end to a variety of circuit design aids programs on a variety of different computers. As such, it is supporting daily production use within Bell Laboratories, primarily at the Holmdel and Merrimack Valley locations. At the same time, however, it is a growing development system. Changes are constantly being made, either as new features are added

* UNIX is a trademark of Bell Laboratories.

or in response to user feedback. This paper first discusses the often conflicting needs of a production and development system. The various methods used to resolve these conflicts and better control the DWB production system are then described. These include the use of a "version control" system, the SCCS[3] and make[4] utilities, and shell procedures.[5]

II. THE NEEDS OF A PRODUCTION SYSTEM

2.1 Reliability

One of the most important factors in a successful production software system is its reliability. Users quickly become discouraged when frequent errors prevent them from working effectively, especially when either the users or the system is new. Many of these errors are due to software bugs, both in original code and in additions and modifications to it. Of course, it is impossible to write error-free software, and programmers can *never* predict all the uses and misuses their programs will suffer. There must be some way, however, for programmers to ease a new program into the system, reducing the number of errors which users experience and which deteriorate their confidence in the production system.

2.2 Maintainability

The operation and maintenance of a production system should be made as automatic as possible, to eliminate the human element which invariably is error-prone and unreliable. All aspects of the system should be cleanly organized and completely documented, facilitating maintenance and reducing the learning curve for new development and maintenance personnel.

2.3 Statistics

Usage and accounting information are required for the management of any production system. The same information can be used to provide feedback information to users on their performance, or to flag those users who are having trouble and need special assistance.

2.4 Responsiveness to users

User complaints, suggestions and requests should be considered extremely important input necessary for the long-range success of a production system. This requires an efficient and effective line of communication between users and developers. User comments should be acknowledged and acted upon as quickly as possible to maintain the users' confidence in the system. If an immediate solution to a problem is impossible or impractical, a proposal for solution and timetable for implementation should be formulated to show that something is being done.

Immediate solutions to user problems usually require software modifications which, as we have noted, sometimes introduce new problems. However, we still want to be able to make changes which can be immediately available for those production users who need them.

III. THE NEEDS OF A DEVELOPMENT SYSTEM

3.1 Control of changes

The software of a development system undergoes constant change. These changes must, first of all, be easy to make. Programmers should not have to wade through complicated procedures to modify their software. On the other hand, the changes must be controlled. Problems are bound to arise if programmers are not aware of the changes being made by others on a common system.

3.2 Statistics

The utilization of various parts of the system can affect where further development efforts should be aimed. If a particular aspect of the system is being heavily used, it may be advantageous to expand the system's capabilities in that area. Also, if a large amount of computer time is being spent in certain programs, it may be appropriate to optimize those programs.

3.3 Documentation

In a constantly evolving development system, all changes must be properly recorded and documented. However, recording and documenting every single change a programmer made to his software would be definite overkill. There should be a happy medium in fulfilling the documentation requirements of a system without overburdening the programmers with unnecessary red tape.

IV. THE PRODUCTION ENVIRONMENT OF DWB

The design and implementation of the production environment of DWB has taken into account all the requirements described above for production and development systems.

4.1 Version control

Many problems outlined above are the result of having a single copy of a software system which must fulfill a variety of functions. These problems would be alleviated by having several complete copies of the system which could coexist and yet be independent, so that changes in one copy, or version, would not affect other versions. This is the basic premise behind the "version control" system which was developed and implemented for DWB.

There are currently a maximum of five separate versions of DWB

maintained by our "version control" system. These are named OLD, CURRENT, RELEASE, TEST, and DEBUG.

4.1.1 The DEBUG version

This version of DWB is available for experimentation and debugging by programmers, both in the development of new modules and the changing of old ones. Since programmers are free to make any changes they want to, this version may not be fully operational at times. Good communication and coordination are necessary between programmers to avoid conflicts in the use of this version.

4.1.2 The TEST version

This version of DWB is intended to contain new features and modifications which have been debugged by programmers. As we have discussed above, new software often results in new bugs, and is therefore too unreliable to incorporate immediately in a production system. The TEST version is intended for users who require or want to try out new features as soon as possible. These users should be relatively "friendly," since they are bound to experience problems. The feedback obtained from these people enables most bugs to be eliminated from the software before it is incorporated and used in the main production system.

4.1.3 The RELEASE version

Eventually, after a number of significant changes and additions have been sufficiently tested by "friendly" users in the TEST version, we are ready to issue a new "release" of DWB. To further insure the reliability of the new release before it is issued, we want to perform regression testing to verify that everything works the way it used to work. This regression testing usually requires a week or two to complete, during which time further changes can be made to the TEST version as development continues. Therefore, when we decide to make a new release, a "frozen" copy of the TEST version is made and called the RELEASE version. This version is only accessible to DWB system personnel, and exists only as long as the new release is undergoing regression testing. When testing is finished, the RELEASE version is made into the CURRENT version.

4.1.4 The CURRENT version

This version of DWB is the main production system, provided as the default for most users. This version is very stable and hence more reliable, since changes are usually made only by new releases or to correct serious bugs.

4.1.5 The OLD version

This version of DWB is a copy of the CURRENT version which existed before the latest release. If unforseen problems develop with a new release, users can still access the OLD version as a backup while repairs are made.

The most important feature of the version control system is that all versions are *simultaneously* available, without any overhead or recompilation required by users or systems personnel. All versions work off the same user data base, so that a user can switch from, say, the CURRENT to the TEST version without losing any files or having to redo anything. This capability is a direct result of our conscious effort to design DWB software in a totally upward compatible manner.

4.2 Version control operations

Each module or program of DWB is treated independently in the version control system. Although various programs may depend closely upon one another and should be treated together, these inter-program dependencies are not automatically handled. Therefore, to describe the operations required to maintain and propagate changes through the version control system, it is only necessary to consider a single module. To facilitate the discussion, the following notation is adopted to define the present state of a particular module within the versions:

M = Module name
I = Issue number of module
Io = I of module in OLD version
Ic = I of module in CURRENT version
Ir = I of module in RELEASE version
It = I of module in TEST version
Id = I of module in DEBUG version

Present state of module M is M (Io,Ic,Ir,It,Id).

The issue number is similar to the SCCS IDentification string (SID) and represents a unique instance of a given module. Whenever a module is altered in any way, the changed module is assigned a new issue number. For example, the following version state expression:

edit (5,5,6,6,6)

indicates that the OLD and CURRENT versions of DWB contain issue 5 of the edit module, while the RELEASE, TEST, and DEBUG versions contain issue 6. Since the RELEASE version may not (and usually does not) exist, its entry in the state expression may be empty:

edit (5,5,,6,6)

4.2.1 Using the DEBUG version

The DEBUG version is available to programmers for experimentation and testing. Making a change to the DEBUG version of the edit module would correspond to the following transition in state expressions:

$$\text{edit (5,5,,5,5)} \rightarrow \text{edit (5,5,,5,6)}$$

Making multiple changes in the DEBUG version of that module could be expressed as multiple changes in the issue number, as in the following:

$$\text{edit (5,5,,5,6)} \rightarrow \text{edit (5,5,,5,7)}$$

However, for simplicity multiple changes are represented as a single change in issue number. This means that the DEBUG version issue number can only differ by one from the TEST version issue number.

4.2.2 Installing the TEST version

When the programmer is satisfied with changes made to a particular module of the DEBUG version, that module can then be installed in the TEST version. This corresponds to te following transition in state expressions:

$$\text{edit (5,5,,5,6)} \rightarrow \text{edit (5,5,,6,6)}$$

At the same time, the "program librarian" manually records a description of the changes that have been made in the module. This is used later to properly update the system documentation and on-line tutorials. The TEST version will now contain the new module which can be tried by "friendly" users. Notice that further changes can be made in the DEBUG version and again installed in the TEST version, as illustrated by the following transitions:

$$\text{edit (5,5,,6,6)} \rightarrow \text{edit (5,5,,6,7)}$$

$$\text{edit (5,5,,6,7)} \rightarrow \text{edit (5,5,,7,7)}$$

4.2.3 Making a new release

When enough changes have been incorporated in the TEST version to warrant a new release, the RELEASE version is created. This corresponds to the following transition:

$$\text{edit (5,5,,6,6)} \rightarrow \text{edit (5,5,6,6,6)}$$

which is performed for every module of DWB. At this point, the RELEASE version is an exact copy of the TEST version, to be used for complete regression testing. Notice, however, that development can

continue unabated in the TEST and DEBUG versions, as reflected by these transitions:

edit (5,5,6,6,6) → edit (5,5,6,6,7)

edit (5,5,6,6,7) → edit (5,5,6,7,7)

When regression testing has been completed on the RELEASE version, it is ready to become the new CURRENT version. At the same time, the CURRENT version must become the OLD version, and the OLD version must be removed. The RELEASE version is also deleted, having become the CURRENT. These changes are reflected in the following transition:

edit (5,5,6,7,7) → edit (5,6,,7,7)

which would be performed for every module of DWB. If the OLD and CURRENT versions of a particular module, say the acquire program, were different, the transition would look like this:

edit (5,6,7,8,8) → edit (6,7,,8,8)

4.3 Source and executable files

The version state expression which defines the state of a particular module is actually a simplification, since programs which are written in *C* consist of both source and executable files which are all maintained under the version control system. Hence, the expression

edit (5,6,,7,7)

really represents all source and executable files of the edit module:

edit.c (5,6,,7,7) and edit.x (5,6,,7,7)

The interrelationship of source and executable files and the fact that their state expressions should be identical is not explicitly enforced in the current implementation of the version control system, and must be handled manually. This problem does not exist for shell procedures, in which the source and executable files are one and the same.

4.4 Simplification of the state expression

The most important fact communicated by the state expressions of DWB modules is the nature of the differences between the versions. For example, the state expressions

edit (5,6,,6,6) and acquire (5,6,,8,9)

indicate the CURRENT and TEST versions of the edit module are identical, while the TEST version acquire module is different from the CURRENT version. In fact, the DEBUG and TEST versions of the acquire module are different, indicating that someone is actively experimenting

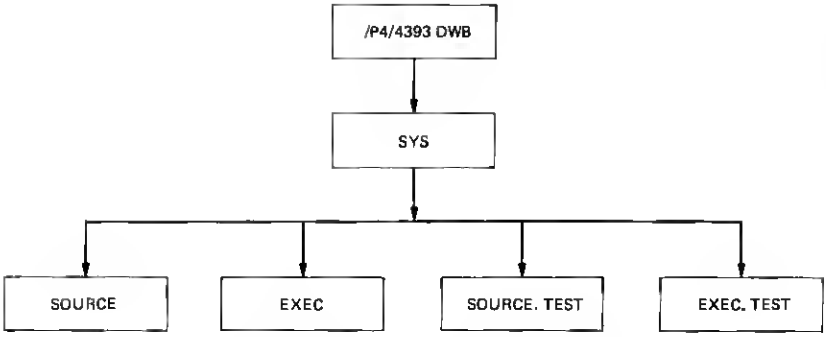

Fig. 1—Directories for CURRENT and TEST.

with that program. To bring out these facts more clearly, the state expressions can be simplified by assigning the letters O, C, T, and D (corresponding to OLD, CURRENT, TEST, and DEBUG) to the issue numbers in the expression as follows:

edit (O,C,,C,C) and acquire (O,C,,T,D)

Notice that an issue number which appears in more than one version takes on the letter corresponding to the leftmost version in the group. Using this notation, it is more clearly shown which modules are common or different between versions.

4.5 Implementation of version control

4.5.1 Version storage

The implementation and operation of the version control system relies heavily on the directory structure in which the source and executable files are stored. This structure is illustrated in part in Fig. 1. Each version of DWB has a directory for its source files and its executable files or programs. Shown in Fig. 1 are the directories for the CURRENT and TEST versions. Naturally, there are equivalent directories for the OLD, RELEASE, and DEBUG versions. Each executable directory contains a complete copy of every program required to run DWB.

4.5.2 Version linkage

Consider the situation expressed by the following:

edit (O,C,,T,D)

This indicates that each version contains a different version of the edit module, and hence there would be different source and executable files in each version directory. Now consider the following situation:

edit (O,O,,T,D)

In this case, the same edit module exists in the OLD and CURRENT version, and hence identical source and executable files would exist in the directories for these versions. To eliminate the wasted space in situations such as this, source and executable files can be linked to several directories. The arrangement of executable files for the situation above is shown in Fig. 2. The UNIX ln command performs this linking function, allowing a file to be stored only once and yet appear in more than one directory. All the operations described in Section 3.2 above (debugging, installing test modules, etc.) must take into account and manipulate these links.

4.5.3 Executing a particular version

DWB uses the UNIX $PATH shell variable to determine the version under which each user is executing. The value of $PATH is a list of the directories in which to search for executable programs. For example, if a user's $PATH had the value

```
:/p4/4393dwb/sys/exec:/bin:/usr/bin
```

all DWB programs would come from the /p4/4393dwb/sys/exec directory, and hence from the CURRENT version. In order for the user to execute the TEST version, the value of $PATH need only be changed to

```
:/p4/4393dwb/sys/exec.test:/bin:/usr/bin
```

so that all programs will come from the /p4/4393dwb/sys/exec.test directory, and hence from the TEST version.

In actuality, DWB users know nothing about the shell variable $PATH. Its value is determined by the DWB main program which the user invokes by typing dwb. The argument given to the program, if any, indicates the desired version. For example, to get the TEST version, the user types dwb test. If no argument is specified, the CURRENT version is used. Since each person logged onto a UNIX system has his own $PATH, each person can access a different version of DWB at the same time.

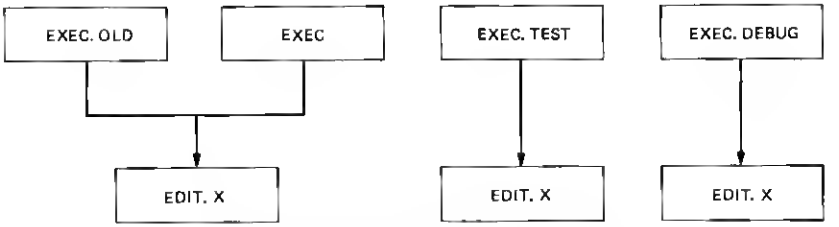

Fig. 2—Executable files for version linkage.

New users have a problem in accessing DWB for the first time, since the program dwb in directory /p4/4393dwb/sys/exec cannot be found by their initial $PATH, which is:

:/bin:/usr/bin

Therefore, to access DWB for the first time, users must type /p4/4393dwb/sys/exec/dwb. At this point, DWB detects that the user is logging in for the first time, and changes his .profile to set the initial $PATH to:

:/bin:/usr/bin:/p4/4393dwb/sys/exec

(The .profile is a set of commands which is executed every time the user logs onto the UNIX system.) From then on, the user need only type dwb, since the proper directory is in the initial $PATH. Notice that we are using the $PATH for two purposes. The user's initial $PATH is used to find the program dwb. Once within this program, the $PATH is changed to reflect the desired version, so that all subsequent programs come from the proper version directory.

The ability to change the $PATH shell variable and, in effect, easily alter the environment in which a user operates was the main ingredient which allowed our implementation of the version control system. This is just one more example of the many simple yet powerful capabilities of the UNIX operating system upon which one can build.

4.6 Compilation control

One of the nice things about the UNIX operating system is the low overhead associated with the initiation of a process. This makes it convenient to write and use small, general-purpose utility programs, in essence creating new commands with which to program in shell procedures. The UNIX "pipe" operation, in which the output of one program is fed as input to another, provides a convenient mechanism for combining such programs. Therefore, the design of a well-structured and modular software system (in which category we naturally place DWB) inevitably contains a large number of individual programs. In developing and maintaining such a system, the problems of managing a large number of source files and the corresponding executable programs quickly become apparent. In particular, it becomes tedious to specify and perform the repeated compilations necessary during development. Fortunately, the UNIX operation system again comes to the rescue with a utility called make. This utility enables automatic recompilation of programs. To use it, you must specify the actions to be taken in compilation, as well as the file dependencies (i.e., which source files, object files, and libraries are required to recompile a particular program). Based on the dependencies, make determines

when a program should be recompiled, and then performs the recompilation. The logic is simple: if a source or object file has changed, all programs depending on that source or object file must be recompiled.

DWB uses the make utility for all its C programs. This use was complicated, however, by the implementation of the version control system described above. It was necessary to write a preprocessor to conveniently specify the recompilation of a particular version of a particular program. For example, to recompile the TEST version of the "edit" program, the source file edit.c from the sys/source.test directory must be used, and the compiled program edit.x must be placed in the sys/exec.test directory. The make utility provides a macro definition capability which helps in this regard. By defining the source and executable directories as macros, the definition of compilation dependencies (the "makefile") can be the same for every version. The preprocessor determines which version is desired by looking at the current directory, and sets the macro values accordingly. For example, to compile the TEST version of the edit program, you would only have to go into the sys/source.test directory and issue the command make.x edit. The preprocessor (make.x) would set the directory macros to sys/source.test and sys/exec.test and then call make to do the recompilation.

The use of the make utility has been invaluable in the development of DWB. A situation in which its use was particularly helpful involved the tutorial mechanism, which provides the on-line documentation via a series of text files. The initial implementation required a rather convoluted compilation procedure in which programs were compiled to make programs that created other programs which had to be compiled. The use of make made the entire procedure automatic, performing only those recompilations necessary with each change. This was quite important, since the tutorial mechanism was included (as a library routine) in a large number of programs. Fortunately, this implementation has been changed to a much more flexible system which is driven directly from the text files and does not require multiple levels of compilation.

4.7 Source code control

The existence of a large number of programs also presents problems in maintaining, changing, and documenting the numerous source files. As usual, UNIX software provides some assistance in the form of a Source Code Control System (SCCS). This system organizes the storage and changing of any type of source file. Each change to a source file is recorded under SCCS as a "delta" and is identified by a version number. Prior versions of a particular source file can be recreated at any time merely by specifying the appropriate version number. SCCS provides

for the storing of various information about each change, such as the programmer's name and the reasons for the change. This organization of information is invaluable in providing and updating system documentation.

Because of the version control mechanism and the organization it provides, we have chosen *not* to use SCCS to record every change to every program. Instead, only the CURRENT versions of DWB are stored under SCCS. This includes all C programs, shell procedures, utility programs, and data files needed for a complete running system. Every time a new issue (and hence a new CURRENT version) of DWB is released, the program and data files stored under SCCS will be updated. This use of SCCS enables us to retrieve a complete running copy of any previous CURRENT version of DWB.

When a new issue of DWB is released, we want to make sure that all documentation is up to date; that is, all changes which have been made since the last issue are reflected in the system documentation and online tutorials. The primary means of doing this is to gather together all the descriptions of changes which have been obtained by the librarian in the installation of modules into the TEST version. These descriptions must then be incorporated into all formal system documentation. The secondary means of making sure that all changes have been documented involves the direct comparison of the new issue with the last issue. Since every file of the last issue of DWB is stored in SCCS, this is no problem. Shell procedures have been written which reconstruct, in temporary directories, the files of the last issue of DWB. These are then compared, one at a time via the UNIX diff utility, with the files of the new issue. Any differences are automatically flagged, indicating the need for a change in the documentation. Naturally, most of the differences will already be covered by the descriptions of changes incorporated in the TEST version. However, by doing a complete comparison of the new and old issues of DWB, we are assured of catching and properly documenting all modifications.

4.8 Statistics

DWB uses a generalized logfile mechanism to record the usage of the various programs in the system. Each circuit has its own logfile which records the activity in that circuit. A system logfile records all logins. The statistics which can be obtained from these files are important in monitoring the usage of the production system to find problem areas. Problems can be caused by software bugs, which need to be corrected, or by the improper use of DWB programs, indicating that users need more help or that the programs must be redesigned for easier use.

A major advantage of the logfile approach is that the type of statistics which can be gathered is not fixed, but can be changed

dynamically. This is because the logfile contains a complete record of all activity, the summarizing and condensation of which is done by report generating utilities (see the next section) and *noi* within the logfile itself.

4.9 Maintenance and operating procedures

A variety of functions must be performed in daily and periodic maintenance of DWB production use as well as in support of development efforts. These require the manipulation of large numbers of files and directories, and if done manually would be extremely tedious and error-prone. Therefore, most of these maintenance and librarian functions have been implemented as shell procedures which are reliable and easy to use. This significantly reduces the knowledge and training required for support of DWB. As an example, the following dialog illustrates the use of the librarian utility which allows the initialization of the DEBUG version of a particular program:

```
*—* DEBUG INITIALIZATION*—*
Is the module a C program or Shell? (c or s) →c
Enter the names of all source files involved (ending with "end")
→ edit.c
*INITIALIZED*
→ end
Enter the names of ALL dependent executable modules (ending with
"end")
→ edit.x
*INITIALIZED*
→ end
*** Thank you, come again! ***
```

In this utility, a check is first made to see if the program is already initialized for debugging. If not, the links from the TEST version to the DEBUG version for that program (both source and executable files) are removed, and separate copies of the TEST version files are placed in the DEBUG version. The mode of the files is then changed to allow the programmer to edit them.

Some of DWB's utilities are designed to run automatically. For example, a cleanup routine is invoked every morning at 3:00 a.m. to find and delete temporary and other garbage files. This utility also looks for inactive circuits for which data base storage can be freed.

Statistics utilities allow the scanning, collection, and summarizing of the circuit and system logfiles. This can be done on a per-circuit, per-project, per-user, and systemwide basis. UNIX utilities, such as grep and wc, are extremely useful in helping to collect such statistics. For example, the total number of logins for a particular circuit can be

found with the following command (where cktlog is the usage logfile for the circuit of interest):

```
grep login cktlog | wc −1
```

The grep program picks out every line from the cktlog file which contains the word login. These lines are then counted by the wc (word count) program.

All the operations required for manipulating the version control system are controlled by shell procedures. These include the initialization of the DEBUG version (illustrated above), the installation of changes into the TEST version, the creation of the RELEASE version, and the installation of a new issue of DWB. A procedure called *modchart* produces a chart which illustrates the current status of every DWB program, indicating the differences between the programs of each version. This chart is basically a listing of the simplified state expression for each program. Also controlled by shell procedures is the updating of the SCCS whenever a new issue of DWB is released.

4.10 Security

The only means of providing file security within the UNIX operating system is through the file protection modes. There are three levels of file access modes. One specifies the access permissions for the owner of the file, the second specifies the access permissions for the group owner of the file, and the third specifies the access permissions for everyone else. DWB uses the group permissions to control access to each project, i.e., each DWB project has a group associated with it, and only that group can work on that project.

Passwords for remote computers are protected on DWB through the use of the crypt command, which performs data encryption based on a "key." The "key" is the user's DWB password, which is used to encrypt the user's other passwords such as the IBM TSS or TSO passwords. Whenever one of these is required, the user is asked to enter his DWB password which is then used to decrypt the desired password. This provides security even from system personnel, since the DWB password is never stored (even in a temporary file). The only capability permitted to DWB system personnel is the complete removal of all the user's passwords, in case the DWB password is forgotten. Verification of the DWB password is accomplished through the storage of an encrypted "constant" string. When the user enters his DWB password, it is used to encrypt the "constant" string, which is compared against the stored encryption of that string.

V. SUMMARY

The production environment of DWB resolves many conflicts between a development and production system. This environment suc-

ceeds in providing a reliable system for production use, as well as a convenient system for debugging and the smooth installation of new features. The version control system was instrumental in providing this versatility, and the features of the UNIX operating system provided a rich pool of resources from which to draw. The overall result is an environment in which work can be productive and enjoyable.

REFERENCES

1. K. Thompson and D. M. Ritchie, "The UNIX Time-Sharing System," B.S.T.J., *57*, No. 6 (July–August 1978), pp. 1905–1930.
2. T. A. Dolotta, R. C. Haight, and J. R. Mashey, "The Programmer's Workbench," B.S.T.J., *57*, No. 6 (July–August 1978), pp. 2177–2200.
3. M. J. Rochkind, "The Source Code Control System," IEEE Trans. on Software Engineering, *SE-1* (December 1975), pp. 364–370.
4. S. I. Feldman, "Make—A Program for Maintaining Computer Programs," Software—Practice and Experience, *9* (April 1979), pp. 255–265.
5. S. R. Bourne, "The UNIX Shell," B.S.T.J., *57*, No. 6 (July–August 1978), pp. 1971–1990.